

# `efit2dg2d`

November 9, 2017  
9:59

## Contents

<b>1</b>	<b>Program for using EFIT equilibrium to set up <i>definegeometry2d</i> input file</b>	<b>1</b>
<b>2</b>	<b>Get nodes from flux surface contours</b>	<b>15</b>

## 1 Program for using EFIT equilibrium to set up *definegeometry2d* input file

This is a general purpose replacement for the IDL routine used to set up main chamber geometries for gas puff imaging studies. In such cases, the region of interest is localized to near the tokamak midplane so that the complexity of the full magnetic geometry does not enter. All that is needed instead is a rectangular area, extended over some range of toroidal angles, and a set of flux surface contours onto which the plasma parameters can be mapped. The box is presently specified via the macros *rbox\_min*, *rbox\_max*, *zbox\_min*, and *zbox\_max*, although these could easily be made input parameters or read from a file. The equilibrium data is read from an EQDSK “g” file, such as produced by the EFIT code, the name of which is specified as a command line argument.

The resolution of the geometry is controlled by the hardwired parameters:

*nrbox* Number of points along the top and bottom of the rectangle.

*nzbox* Number of points along the sides of the rectangle.

*n\_cont\_in* Number of contours inside the separatrix.

*n\_cont\_out* Number of contours outside the separatrix.

*r\_dtheta* Target angular distance between points along the flux surfaces.

The sizes of the triangles that are produced by *definegeometry2d* are determined by *nrbox*, *nzbox*, and *r\_dtheta*. The radial resolution is controlled by *n\_cont\_in* and *n\_cont\_out*; e.g., to resolve a steep pedestal, one might need a larger value of *n\_cont\_in*.

The code produces output files:

*wallfile* List of “walls” generated from the box edges and the flux surface contours suitable for input to *definegeometry2d*.

*polygons\_dg2d.in* Corresponding polygon specifications that can be pasted into the construction section of a *definegeometry2d* input file.

*polygons.txt* Explicit *R*, *Z* listing of these polygons suitable for external plotting with the *read\_poly.py* script.

*stratum\_psi* Listing of the “stratum” values used for each of the polygons in *polygons\_dg2d.in* together with the corresponding poloidal flux, normalized to the separatrix value. These data can be read in by a user-defined *get\_n\_t* routine (via *usr2dplasma.web*) compiled into *defineback* and which can then map the plasma parameters onto the polygons.

Note that this routine does not provide a complete input file for *definegeometry2d*. At a minimum, the user needs to provide the preamble which specifies the geometry symmetry, bounds (which should extend beyond the coordinates *rbox\_min*, etc. used to specify the rectangle here), and the wallfile. The construction section also needs additional solid or exit polygons fill the gap between the rectangle used here and that represented by the bounds keyword. In practice, one also needs to provide polygons representing the gas source, vessel walls, and the intervening vacuum and / or plasma regions. Since these are machine and problem dependent, their implementation is left to the user.

```
$Id: $

"efit2dg2d.f" 1≡
@m FILE 'efit2dg2d.web'

@m open_file(aunit, aname)
open(unit = aunit, file = aname, status = 'old', form = 'formatted', iostat = open_stat)
assert(open_stat ≡ 0)

@m increment_num_nodes num_nodes++
dim_nodes = max(dim_nodes, num_nodes)
var_realloc(nodes)
var_realloc(node_type)
var_realloc(node_element_count)
dim_nodes = mem_size(dim_nodes)

@m increment_num_walls num_walls++
dim_walls = max(dim_walls, num_walls)
var_realloc(wall_nodes)
var_realloc(wall_elements)
var_realloc(wall_psi_contour)
var_realloc(wall_segment_count)
dim_walls = mem_size(dim_walls)

@m increment_ext_lim_m ext_lim_m++
dim_ext = max(dim_ext, ext_lim_m)
var_realloc(ext_lim_r)
var_realloc(ext_lim_z)
var_realloc(ext_lim_wall)
var_realloc(ext_lim_index)
dim_ext = mem_size(dim_ext)

@m n_opt 7
@m i_newt_max 50
@m wall_unused 0
@m wall_used_once 1
@m wall_used_twice 2
```

The Main Program.

```
"efit2dg2d.f" 1.1 ≡
@m NSTX_ENDD 0
@m NSTX_GPI 1
@m CMOD_GPI 2
@m APP CMOD_GPI

program efit2dg2d
    define_dimen(rb_ind, nrbox)
    define_dimen(zb_ind, nzbox)
    define_dimen(cont_ind, n_cont)
    define_dimen(ext_lim_ind, dim_ext)
    define_varp(rbox, FLOAT, rb_ind)
    define_varp(zbox, FLOAT, zb_ind)
        /* The wall structures are patterned exactly after the ones in definegeometry2d. */
    define_dimen(node_ind, dim_nodes)
    define_dimen(wall2d_ind, dim_walls)
    define_dimen(wall_used_ind, 0, num_walls)
    define_varp(nodes, FLOAT, g2_xz_ind, node_ind)
    define_varp(node_type, INT, node_ind)
    define_varp(node_element_count, INT, node_ind)
    define_varp(wall_nodes, INT, g2_points_ind0, wall2d_ind)
    define_varp(wall_elements, INT, g2_points_ind, wall2d_ind)
    define_varp(wall_psi_contour, INT, wall2d_ind)
    define_varp(wall_segment_count, INT, wall2d_ind)
    define_varp(wall_used, INT, wall_used_ind)
    define_varp(psi_contours, FLOAT, cont_ind)
    define_varp(ext_lim_r, FLOAT, ext_lim_ind)
    define_varp(ext_lim_z, FLOAT, ext_lim_ind)
    define_varp(ext_lim_wall, INT, ext_lim_ind)
    define_varp(ext_lim_index, FLOAT, ext_lim_ind)
    define_varp(ext_lim_used, INT, ext_lim_ind)
    define_varp(ext_lim_tempf, FLOAT, ext_lim_ind)
    define_varp(ext_lim_tempi, INT, ext_lim_ind)

    implicit none_f77
    ef_common // Common
    implicit none_f90

integer nargs, nrbox, nzbox, ir, iz, im, direction, is,      /* Local */
       dim_nodes, num_nodes, dim_walls, num_walls, n_cont_in, n_cont_out, n_cont, ic, imp1, closer,
       ext_lim_m, dim_ext, first_point, num_poly, next_start, iwall, im_next, num_poly_wall
integer poly_wall2
real rbox_min, rbox_max, zbox_min, zbox_max, drbox, dzbox, dr, r, z, psi, psi_min, psi_max,
     psi_value, delta_psi, lim_psi_min, lim_psi_max, poly_psi, psilim
real segment_2, x_psi_2
logical tracked, polygons_done
character*FILELEN g_file

integer finished, success, convergence_error_count,      /* Get Nodes */
       itr, nseg, nseg_p
real r_dtheta, psi_err, max_err, dist
real x_old_2, x_new_2, lines_2, x_int_2
```

```

external psi_interp, wall_cross // External
integer wall_cross
real psi_interp

⟨ Memory allocation interface 0 ⟩
sy_decls
st_decls

declare_varp(rbox)
declare_varp(zbox)

declare_varp(nodes)
declare_varp(node_type)
declare_varp(node_element_count)

declare_varp(wall_nodes)
declare_varp(wall_elements)
declare_varp(wall_psi_contour)
declare_varp(wall_segment_count)
declare_varp(wall_used)

declare_varp(psi_contours)

declare_varp(ext_lim_r)
declare_varp(ext_lim_z)
declare_varp(ext_lim_wall)
declare_varp(ext_lim_index)
declare_varp(ext_lim_used)
declare_varp(ext_lim_tempf)
declare_varp(ext_lim_tempi)

nargs = arg_count()
if (nargs ≠ 1) then
    assert('Command_line_must_specify_the_name_of_the_EFIT_g-file' == '')
end if
call command_arg(1, g-file)

call init_psi_interp(g_file) /* This prototype attempts to replicate an ENDD case. Now narrow
   the focus down to a rectangular box. These values are for the low-field side, midplane of NSTX.
   While the specification of the corners of the box could be done interactively, alot of the code
   that follows later makes specific assumptions about the possible paths of the contours through
   the box. */

@#if (APP ≡ NSTX_ENDD)
    rbox_min = const(1.2)
    rbox_max = const(1.555)
    zbox_min = const(-0.40)
    zbox_max = const(0.54)
@#elif (APP ≡ CMOD_GPI)
    rbox_min = const(0.82)
    rbox_max = const(0.926)
    zbox_min = const(-0.12)
    zbox_max = const(0.12)
@#endif
    nrbox = 35
    nzbox = 75
    drbox = (rbox_max - rbox_min) / (areal(nrbox - 1))
    dzbox = (zbox_max - zbox_min) / (areal(nzbox - 1))

```

```

var_alloc(rbox)
var_alloc(zbox)

do ir = 1, nrbox
  rbox_ir = rbox_min + areal(ir - 1) * drbox
end do

psi_min = const(1., 16)
psi_max = const(-1., 16)
do iz = 1, nzbox
  zbox_iz = zbox_min + areal(iz - 1) * dzbox
  do ir = 1, nrbox
    psi_value = psi_interp(rbox_ir, zbox_iz, 0, 0)
    if (psi_value < psi_min)
      psi_min = psi_value
    if (psi_value > psi_max)
      psi_max = psi_value
  end do
end do
assert(psi_min < psi_max) /* Hard wire these for now, but expect that they will likely be input somehow. The extra "1" is for the separatrix. Note that we are assuming that the separatrix is at  $\psi = ef\_psilim$ . We are proceeding under the assumption that there are no X- or O-points inside the box. In that case, the maximum and minimum  $\psi$  values will be on the boundary of the box. For this reason, the contours at the maximum and minimum values will be null. To bypass them, add 1 to the denominator when computing  $\delta\psi$ . */

n_cont_in = 25
n_cont_out = 25
n_cont = n_cont_in + n_cont_out + 1
var_alloc(psi_contours)
delta_psi = (ef_psilim - psi_min) / areal(n_cont_in + 1)
do ic = 1, n_cont_in
  psi_contours_ic = psi_min + areal(ic) * delta_psi
end do
psi_contours_{n_cont_in+1} = ef_psilim
delta_psi = (psi_max - ef_psilim) / float(n_cont_out + 1)
do ic = 1, n_cont_out
  psi_contours_{ic+n_cont_in+1} = ef_psilim + float(ic) * delta_psi
end do /* Convention used here for the "limiter" is that  $ef\_lim\_msegments$  is the number of segments. The first point in each segment has the same index as the segment. There are then  $ef\_lim\_msegments + 1$  total points. But, that last point is the same as the first one and is not explicitly stored. This convention has been borrowed from the XGC-0 "limiter" routines since this procedure makes use of other subroutines and techniques from there. In contrast, the indexing of the "walls" ( $\psi$  contours) mirrors that used in definegeometry2d: the points or nodes on a wall are instead numbered beginning with 0. There are  $wall\_segment\_count_{iwall}$  segments in wall  $iwall$  and  $wall\_segment\_count_{iwall} + 1$  nodes. The last node is  $wall\_nodes_{iwall, wall\_segment\_count_{iwall}}$ . */

ef_lim_msegments = 2 * (nrbox - 1) + 2 * (nzbox - 1)
var_alloc(ef_lim_r)
var_alloc(ef_lim_z)
var_alloc(ef_lim_psi)
var_alloc(ef_lim_cross_times)
var_alloc(ef_lim_cross_node)
var_alloc(ef_lim_cross_wall)
lim_psi_min = const(1., 16)

```

```

lim_psi_max = const(-1., 16)
im = 0
ir = 1
do iz = 1, nzbox
    im++
    ef_lim_r_im = rbox_ir
    ef_lim_z_im = zbox_iz
    ef_lim_psi_im = psi_interp(rbox_ir, zbox_iz, 0, 0)
    if (ef_lim_psi_im < lim_psi_min)
        lim_psi_min = ef_lim_psi_im
    if (ef_lim_psi_im > lim_psi_max)
        lim_psi_max = ef_lim_psi_im
    ef_lim_cross_times_im = 0
    do ic = 1, ef_cross_max
        ef_lim_cross_node_im,ic = int_undef
        ef_lim_cross_wall_im,ic = int_undef
    end do
end do
iz = nzbox
do ir = 2, nrbox
    im++
    ef_lim_r_im = rbox_ir
    ef_lim_z_im = zbox_iz
    ef_lim_psi_im = psi_interp(rbox_ir, zbox_iz, 0, 0)
    if (ef_lim_psi_im < lim_psi_min)
        lim_psi_min = ef_lim_psi_im
    if (ef_lim_psi_im > lim_psi_max)
        lim_psi_max = ef_lim_psi_im
    ef_lim_cross_times_im = 0
    do ic = 1, ef_cross_max
        ef_lim_cross_node_im,ic = int_undef
        ef_lim_cross_wall_im,ic = int_undef
    end do
end do
ir = nrbox
do iz = nzbox - 1, 1, -1
    im++
    ef_lim_r_im = rbox_ir
    ef_lim_z_im = zbox_iz
    ef_lim_psi_im = psi_interp(rbox_ir, zbox_iz, 0, 0)
    if (ef_lim_psi_im < lim_psi_min)
        lim_psi_min = ef_lim_psi_im
    if (ef_lim_psi_im > lim_psi_max)
        lim_psi_max = ef_lim_psi_im
    ef_lim_cross_times_im = 0
    do ic = 1, ef_cross_max
        ef_lim_cross_node_im,ic = int_undef
        ef_lim_cross_wall_im,ic = int_undef
    end do
end do
iz = 1
do ir = nrbox - 1, 2, -1

```

```

im++
ef_lim_r_im = rbox_ir
ef_lim_z_im = zbox_iz
ef_lim_psi_im = psi_interp(rbox_ir, zbox_iz, 0, 0)
if (ef_lim_psi_im < lim_psi_min)
    lim_psi_min = ef_lim_psi_im
if (ef_lim_psi_im > lim_psi_max)
    lim_psi_max = ef_lim_psi_im
ef_lim_cross_times_im = 0
do ic = 1, ef_cross_max
    ef_lim_cross_node_im,ic = int_undef
    ef_lim_cross_wall_im,ic = int_undef
end do
end do
assert(im == ef_lim_msegments) /* Check  $\psi$  values on boundary. If these do not match, suggests
the presence of an X- or O-point in the volume. */
assert(lim_psi_min < lim_psi_max)
assert(abs((lim_psi_min - psi_min) / max(lim_psi_min, psi_min)) < epsilon)
assert(abs((lim_psi_max - psi_max) / max(lim_psi_max, psi_max)) < epsilon)
    /* Initialize the extended limiter arrays. */
ext_lim_m = ef_lim_msegments
dim_ext = mem_size(ext_lim_m)
var_alloc(ext_lim_r)
var_alloc(ext_lim_z)
var_alloc(ext_lim_wall)
var_alloc(ext_lim_index)
do im = 1, ext_lim_m
    ext_lim_r_im = ef_lim_r_im
    ext_lim_z_im = ef_lim_z_im
    ext_lim_wall_im = 0
    ext_lim_index_im = areal(im)
end do
dim_nodes = mem_inc
dim_walls = mem_inc
var_alloc(nodes)
var_alloc(node_type)
var_alloc(node_element_count)
num_nodes = 0
var_alloc(wall_nodes)
var_alloc(wall_elements)
var_alloc(wall_psi_contour)
var_alloc(wall_segment_count)
num_walls = 0 /* One step closer to the desired functionality: Search the box boundary for the  $\psi$ 
values in psi_contours. When one is found, track it through the box to make the next wall. Add
the crossing points to the boundary to facilitate the subsequent construction of polygons. Note
the separate equality test on the  $\psi$  values: to avoid a duplicate surface, associate with only the
the first point of a wall segment. The cross references here may be confusing, so spell them out
explicitly:
```

*ef\_lim\_psi\_im* Numerical value of  $\psi$  at limiter point *im*, starting point of limiter segment *im*.

$\psi_{contour_{ic}}$  Numerical value of  $\psi$  for contour  $ic$ . The objective of the tracking procedure is to make “walls” out of these contours.

$wall_{\psi\_contour_{iwall}}$  This index into the  $\psi_{contour}$  array provides the  $\psi$  value associated with wall  $iwall$ .

$ef\_lim\_cross\_times_{im}$  Number of times limiter segment  $im$  has been intersected by a “wall” (equivalently, a  $\psi_{contour}$ ).

$ef\_lim\_cross\_wall_{im,ic}$  Is the wall number associated with the  $ic$ -th crossing of limiter segment  $im$ .

$ef\_lim\_cross\_node_{im,ic}$  Is the node number associated with the  $ic$ -th crossing of limiter segment  $im$ .

```

*/
do ic = 1, n_cont
  psi = psi_contours_ic
  do im = 1, ef_lim_msegments
    imp1 = 1 + mod(im, ef_lim_msegments)
    if (((ef_lim_psi_im - psi) * (ef_lim_psi_imp1 - psi) < zero) ∨ (ef_lim_psi_im ≡ psi)) then
      /* Then this psi_contour crosses the limiter here. Need to then see if it has been tracked
       and made into a contour yet. */
      tracked = F
    if (ef_lim_cross_times_im > 0) then
      do ir = 1, ef_lim_cross_times_im
        if (wall_psi_contour_ef_lim_cross_wall_im,ir ≡ ic)
          tracked = T
      end do
    end if
    if (¬tracked) then /* Has not been tracked yet. Need to then do so and follow it. */
      increment_num_walls
      wall_segment_count_num_walls = 0
      wall_psi_contour_num_walls = ic
      segment_1,1 = ef_lim_r_im
      segment_1,2 = ef_lim_z_im
      segment_2,1 = ef_lim_r_imp1
      segment_2,2 = ef_lim_z_imp1
      closer = im
      if (abs(ef_lim_psi_imp1 - psi) < abs(ef_lim_psi_im - psi))
        closer = imp1
      x_psi_1 = ef_lim_r_closer
      x_psi_2 = ef_lim_z_closer
      call psi_solve(psi, segment, x_psi)
      r = x_psi_1
      z = x_psi_2
      increment_num_nodes
      nodes_num_nodes,g2_x = r
      nodes_num_nodes,g2_z = z
      wall_nodes_num_walls,0 = num_nodes
      increment_ext_lim_m
      ext_lim_r_ext_lim_m = r
      ext_lim_z_ext_lim_m = z
      ext_lim_wall_ext_lim_m = num_walls // Positive for wall start
      ext_lim_index_ext_lim_m = areal(im) + sqrt((r - ef_lim_r_im)^2 + (z - ef_lim_z_im)^2) /
        sqrt((ef_lim_r_imp1 - ef_lim_r_im)^2 + (ef_lim_z_imp1 - ef_lim_z_im)^2)
      /* Record the limiter crossing at the starting point of the wall / contour. There is a
       corresponding record at the end of the wall / contour in GetNodes. */
      ef_lim_cross_times_im ++
      assert(ef_lim_cross_times_im ≤ ef_cross_max)
      ef_lim_cross_wall_im,ef_lim_cross_times_im = num_walls
      ef_lim_cross_node_im,ef_lim_cross_times_im = num_nodes
      /* Determine the direction parameter needed by Hager's tracking algorithm. The
       contours are tracked in the direction of  $\nabla\psi \times \hat{\ell}$ . Assuming that the direction traced out
       by the limiter boundary,  $\hat{\ell}$ , is clockwise (which is the sense we want for the polygons in
       the end), then direction is given by the sign of  $-\hat{\ell} \cdot \nabla\psi$ . */
      direction = int(sign(one, -(psi_interp(r, z, 1,

```

```

    0) * (segment2,1 - segment1,1) + psi_interp(r, z, 0, 1) * (segment2,2 - segment1,2)))
assert(direction ≠ zero)
⟨ Get Nodes 2⟩
end if
end if // Check on  $\psi$  being on this limiter segment
end do // Loop over limiter segments
end do // Loop over  $\psi$  contours /* Sort on ext_lim_index. Use ext_lim_used to hold the mapping
for the process. Also need temporary arrays to then apply the mapping to the unsorted arrays.
var_alloc(ext_lim_used)
var_alloc(ext_lim_tempf)
var_alloc(ext_lim_tempi)
do im = 1, ext_lim_m
  ext_lim_usedim = im
end do
call sort2(ext_lim_m, ext_lim_index, ext_lim_used)
do im = 1, ext_lim_m
  ext_lim_tempfim = ext_lim_rim
  ext_lim_tempiim = ext_lim_wallim
end do
do im = 1, ext_lim_m
  ext_lim_rim = ext_lim_tempfext_lim_usedim
  ext_lim_wallim = ext_lim_tempiext_lim_usedim
end do
do im = 1, ext_lim_m
  ext_lim_tempfim = ext_lim_zim
end do
do im = 1, ext_lim_m
  ext_lim_zim = ext_lim_tempfext_lim_usedim
end do
do im = 1, ext_lim_m
  ext_lim_usedim = FALSE
end do /* Make one last wall out of the extended limiter surface. */
increment_num_walls
wall_segment_countnum_walls = ext_lim_m
wall_psi_contournum_walls = int_undef
do im = 1, ext_lim_m
  if (ext_lim_indexim ≡ int(ext_lim_indexim)) then
    /* Need to make nodes out of the original limiter points. */
    increment_num_nodes
    nodesnum_nodes,g2_x = ext_lim_rim
    nodesnum_nodes,g2_z = ext_lim_zim
    wall_nodesnum_walls,im-1 = num_nodes
  else /* Can get the node numbers of the extended limiter points from the wall_nodes using the
fact that these are the first (zeroth) and last nodes on each wall. */
    if (ext_lim_wallim > 0) then
      wall_nodesnum_walls,im-1 = wall_nodesext_lim_wallim,0
    else
      wall_nodesnum_walls,im-1 = wall_nodes-ext_lim_wallim,wall_segment_count-ext_lim_wallim
    end if
  end if
end do /* Since the “walls” are not necessarily closed surfaces, need to explicitly list the last

```

```

point; same as the first one here. */
wall_nodes num_walls,wall_segment_count num_walls = wall_nodes num_walls,0
/* Write out the wallfile. */
open(unit = diskout, file = 'wallfile', status = 'unknown')
write(diskout, '(a)') '#_Total_number_of_walls'
write(diskout, '(i5)') num_walls
write(diskout, '(a)') '#_Number_of_points_in_each_wall'
/* Purposely not using a format here since this is the way this is read in by definegeometry2d. */
write(diskout, *) SP(wall_segment_count im + 1, im = 1, num_walls)
do im = 1, num_walls
  if (im < num_walls) then
    write(diskout,
      '(a,i4,a,1pe14.6,a,0p,f9.6)') '#_Wall', im, ',_psi=', psi_contours wall_psi_contour_im,
      ',_psi/psi_sep,', psi_contours wall_psi_contour_im / ef_psilim
  else
    write(diskout, '(a,i4)') '#_Boundary_wall,', im
  end if
  do ir = 0, wall_segment_count im
    write(diskout, '(1p,2(2x,e14.6))') nodes wall_nodes im,ir,g2_x, nodes wall_nodes im,ir,g2_z
  end do
end do
close(unit = diskout) /* Write out polygons. */
open(unit = diskout, file = 'polygons_dg2d.in', status = 'unknown')
write(diskout, *) '#'
write(diskout, *) '#These are all flux contour polygons'
write(diskout, *) '#' /* A file to connect them to the  $\psi$  values. */
open(unit = diskout + 1, file = 'stratum_psi', status = 'unknown')
write(diskout + 1, *) '_stratum_psi/psi_sep'
/* This one dumps out the polygons as R, Z pairs that can be plotted externally. */
open(unit = diskout + 2, file = 'polygons.txt', status = 'unknown')

var_alloc(wall_used)
do im = 0, num_walls
  wall_used(im) = wall_unused
end do
first_point = 0
im = 1
num_poly = 0
polygons_done = F
do while(¬polygons_done ∨ (first_point ≠ 0))
  if (first_point ≡ 0) then
    first_point = im
    assert(im ≤ ext_lim_m)
    num_poly++
    write(diskout + 2, *) 'Polygon:', num_poly
    next_start = 0
    write(diskout, *) 'new_zone_plasma'
    write(diskout, *) '_new_polygon'
    write(diskout, '(a,i4)') '_stratum', num_poly
    poly_wall_1 = 0
    poly_wall_2 = 0
    num_poly_wall = 0
  end if

```

```

if (ext_lim_wall_im ≡ 0) then /* Add the next segment of the extended limiter. For the
   polygons, could group consecutive points together if this approach proves too messy in
   practice. Note the use of im – 1 since the wall nodes begin numbering at 0. The outer wall
   (num_walls) is closed, so need not worry about distinguishing the first and last points. */
assert (ext_lim_used_im ≡ FALSE)
write (diskout + 2, *) ext_lim_r_im, ext_lim_z_im
write (diskout, '(a,3i4)' )'_____wall_', num_walls, im – 1, im – 1
ext_lim_used_im = TRUE
im ++
if (im ≡ ext_lim_m + 1)
   im = 1
else /* Add a wall to the polygon. Before doing so, search ahead in ext_lim_used for the next
   unused extended limiter segment; this will be the starting point for the next polygon. */
if (next_start ≡ 0)
   im_next = im
do while (next_start ≡ 0)
   im_next = im_next + 1
   if (im_next ≡ ext_lim_m + 1)
      im_next = 1
   iwall = ext_lim_wall_im_next
   if ((ext_lim_used_im_next ≡ FALSE) ∧ (wall_used_iwall ≡ wall_unused)) then
      next_start = im_next
   else
      if (im_next ≡ im) then // Have gone all the way round
         polygons_done = T
         next_start = im_next
      end if
   end if
   end do
if (ext_lim_wall_im ≠ 0) then
   iwall = ext_lim_wall_im
   if (ext_lim_wall_im > 0) then
      do ir = 0, wall_segment_count_iwall
         write (diskout + 2, *) nodes_wall_nodes_iwall,ir,g2_x, nodes_wall_nodes_iwall,ir,g2_z
         if (ir ≡ 0)
            write (diskout, '(a,i4,a)' )'_____wall_', iwall, '_*'
      end do
   else
      do ir = wall_segment_count_iwall, 0, -1
         write (diskout + 2, *) nodes_wall_nodes_iwall,ir,g2_x, nodes_wall_nodes_iwall,ir,g2_z
         if (ir ≡ 0)
            write (diskout, '(a,i4,a)' )'_____wall_', -iwall, '_*'_reverse'
      end do
   end if
   wall_used_abs(iwall) ++
   num_poly_wall ++
   if (num_poly_wall ≤ 2) then
      poly_wall_num_poly_wall = abs(iwall)
   else if (num_poly_wall ≡ 3) then /* Can get 3 walls, but two will have the same  $\psi$  value.
      Verify and throw out the duplicate. */
      if (wall_psi_contour_poly_wall1 ≡ wall_psi_contour_poly_wall2) then
         assert (wall_psi_contour_abs(iwall) ≠ wall_psi_contour_poly_wall2)

```

```

num_poly_wall = 2
poly_wall_num_poly_wall = abs(iwall)
else
  assert((wall_psi_contourpoly_wall1 ≡ wall_psi_contourabs(iwall)) ∨
         (wall_psi_contourpoly_wall2 ≡ wall_psi_contourabs(iwall)))
end if
else
  assert('More than 3 walls!' ≡ ' ')
end if
end if
im = int_lookup(-iwall, ext_lim_wall, ext_lim_m)
assert((im > 0) ∧ (im ≤ ext_lim_m))
ext_lim_usedim = TRUE
im++
if (im ≡ ext_lim_m + 1)
  im = 1
end if // On ext_lim_wall
if (im ≡ first_point) then /* Add this point to close polygon. */
  write(diskout + 2, *) ext_lim_rim, ext_lim_zim
  write(diskout, '(a,3i4)' )'wall', num_walls, im - 1, im - 1
  write(diskout, '(a)' )'triangulate_to_zones'
  if (poly_wall2 > 0) then
    poly_psi = half * (psi_contourswall_psi_contour poly_wall1 + psi_contourswall_psi_contour poly_wall2)
  else
    poly_psi = psi_contourswall_psi_contour poly_wall1
  end if /* For present purposes, actually want the poloidal flux normalized to the limiter /
           separatrix value, not relative. The second value goes with the revised approach, tracking
           XGC, of referring all  $\psi$  values to the magnetic axis. */
  @#if 0
    poly_psi = one + poly_psi / ef_psilim
  @#else
    poly_psi = poly_psi / ef_psilim
  @#endif
  write(diskout + 1, '(2x,i4,2x,f10.7)' ) num_poly, poly_psi
  first_point = 0
  im = next_start
end if
end do // Outer do-while loop
close(unit = diskout)
close(unit = diskout + 1)
close(unit = diskout + 2)
var_free(rbox)
var_free(zbox)
var_free(nodes)
var_free(node_type)
var_free(node_element_count)
var_free(wall_nodes)
var_free(wall_elements)
var_free(wall_psi_contour)
var_free(wall_segment_count)

```

```
var_free(wall_used)
var_free(ext_lim_r)
var_free(ext_lim_z)
var_free(ext_lim_wall)
var_free(ext_lim_index)
var_free(ext_lim_used)
var_free(ext_lim_tempf)
var_free(ext_lim_tempi)

stop
end
```

⟨ Functions and Subroutines 2.1 ⟩

## 2 Get nodes from flux surface contours

This routine and the ones invoked by it have been borrowed from Hager's gengrid2 package, primarily from node3.F90. These have been modified for the simpler box-like geometries anticipated for this code and to use the *bspline90\_22.f90* spline fitting routines.

```

⟨ Get Nodes 2⟩ ≡
  dr = drbox
  r_dtheta = const(3., -3)    // target poloidal distance

  x_old1 = r
  x_old2 = z

  finished = 0
  max_err = zero
  convergence_error_count = 0
  do while(finished ≡ 0)      /* Walk along flux-surface to get next point */
    call get_next_point_tang(x_old, r_dtheta, direction, x_new, dist)
    /* Refine psi value of new point */
    call fsrefine_xgca(x_new, psi, dr, n_opt, psi_err, success)
    if (success ≠ 1)
      convergence_error_count = convergence_error_count + 1
    if (psi_err > max_err)
      max_err = psi_err
    /* Check whether the new point is inside (itr=1) or outside (itr = 0) the boundary. */
    itr = -1
    call wall_check(itr, x_new1, x_new2)
    if (itr < 1) then      /* The wall was crossed;a boundary node has to be added. */
      nseg = wall_cross(x_old1, x_old2, x_new1, x_new2)
      nseg_p = 1 + mod(nseg, ef_lim_msegments)
      /* Find the point of intersection of the crossing segment with the wall segment. The parameter
         alpha represents the fractional distance along the wall segment from point nseg. */
      lines1,1,1 = x_old1
      lines1,1,2 = x_old2
      lines1,2,1 = x_new1
      lines1,2,2 = x_new2
      lines2,1,1 = ef_lim_rnseg
      lines2,1,2 = ef_lim_znseg
      lines2,2,1 = ef_lim_rnseg_p
      lines2,2,2 = ef_lim_znseg_p
      call find_intersection(lines, x_int)
      call psi_solve(psi, lines2,1,1, x_int)
      finished = 1
      x_new1 = x_int1
      x_new2 = x_int2    /* Double check the value of nseg since the final point may not be on the
                                originally identified segment. This just looks at the signs dot-products of the vectors formed
                                by the new point and the limiter points nseg and nseg_p. */
      if ((x_new1 - ef_lim_rnseg) * (x_new1 - ef_lim_rnseg_p) + (x_new2 - ef_lim_znseg) * (x_new2 -
          ef_lim_znseg_p) > zero) then
        if ((x_new1 - ef_lim_rnseg) * (ef_lim_rnseg_p - ef_lim_rnseg) + (x_new2 - ef_lim_znseg) *
            (ef_lim_znseg_p - ef_lim_znseg) > zero) then
          nseg = nseg_p
          nseg_p = 1 + mod(nseg, ef_lim_msegments)

```

```

else
  nseg_p = nseg
  nseg = nseg_p - 1 // mod?
end if
  assert((x_new_1 - ef_lim_r_nseg) * (x_new_1 - ef_lim_r_nseg_p) + (x_new_2 - ef_lim_z_nseg) * (x_new_2 - ef_lim_z_nseg_p) < zero)
end if
end if
increment_num_nodes
nodes_num_nodes,g2_x = x_new_1
nodes_num_nodes,g2_z = x_new_2
wall_segment_count_num_walls ++
assert(wall_segment_count_num_walls ≤ g2_num_points - 1)
wall_nodes_num_walls,wall_segment_count_num_walls = num_nodes

x_old_1 = x_new_1
x_old_2 = x_new_2
end do
ef_lim_cross_times_nseg ++
assert(ef_lim_cross_times_nseg ≤ ef_cross_max)
ef_lim_cross_wall_nseg,ef_lim_cross_times_nseg = num_walls
ef_lim_cross_node_nseg,ef_lim_cross_times_nseg = num_nodes
increment_ext_lim_m
ext_lim_r_ext_lim_m = x_new_1
ext_lim_z_ext_lim_m = x_new_2
ext_lim_wall_ext_lim_m = -num_walls // Negative for wall end
ext_lim_index_ext_lim_m = areal(nseg) + sqrt((x_new_1 - ef_lim_r_nseg)^2 + (x_new_2 - ef_lim_z_nseg)^2) /
  sqrt((ef_lim_r_nseg_p - ef_lim_r_nseg)^2 + (ef_lim_z_nseg_p - ef_lim_z_nseg)^2)

```

This code is used in section 1.1.

Version of the XGC routine *wall\_check* specific for the limiter structure used here. Determines whether the new point is inside (*itr*=1) or outside (*itr* = 0) the boundary.

```

⟨ Functions and Subroutines 2.1 ⟩ ≡
subroutine wall_check(itr, r, z)
  implicit none_f77
  ef_common // Common
  implicit none_f90

  integer itr // Input / output
  real r, z

  integer test_lim // Local
  external pointinpoly // External
  integer pointinpoly

  itr = 1 // Inside the wall
  test_lim = pointinpoly(r, z, ef_lim_r, ef_lim_z, ef_lim_msegments)

  if (test_lim ≤ 0) then
    itr = 0 // Outside the wall
  end if

  return
end
```

See also sections 2.2, 2.3, and 2.4.

This code is used in section 1.1.

Version of the XGC routine *wall\_cross* specific for the limiter structure used here. This is a general routine for identifying the segment of a closed polygon intersected by the input segment. If no intersection is found, the routine returns 0.

The particular application in this code is to determine which segment of the boundary was crossed by a particle, with the crossing have been previously flagged by subroutine *wall\_check*. The first point of the segment is the previous location of the particle. The second is its current location, which should be outside the boundary. Should this segment not intersect the boundary for some reason, the routine will return 0.

*<Functions and Subroutines 2.1> +≡*

```

function wall_cross(r1, z1, r2, z2)
  implicit none_f77
  ef_common // Common
  implicit none_f90

  integer wall_cross // Function
  real r1, z1, r2, z2 // Input

  integer j, jp // Local
  real q_ir, q_iz, delta_q_r, delta_q_z, delta_q_2, p_jr, p_jz, delta_p_r, delta_p_z, delta_p_2, alpha, beta
  q_ir = r1
  q_iz = z1
  delta_q_r = r2 - r1
  delta_q_z = z2 - z1
  delta_q_2 = delta_q_r^2 + delta_q_z^2
  if ((abs(delta_q_r) + abs(delta_q_z)) ≡ zero) then
    assert('Particle takes null step but crossed boundary! ' ≡ ' ')
  end if

  wall_cross = 0
  do j = 1, ef_lim_msegments
    p_jr = ef_lim_r_j
    p_jz = ef_lim_z_j /* Easy enough to allow for polygon to close on itself or not. */
    jp = 1 + mod(j, ef_lim_msegments)
    delta_p_r = ef_lim_r_jp - p_jr
    delta_p_z = ef_lim_z_jp - p_jz
    delta_p_2 = delta_p_r^2 + delta_p_z^2 /* Skip any null segments */
    if ((abs(delta_p_r) + abs(delta_p_z)) > zero) then
      /* The parameter alpha is the distance of the point of intersection (of the two segments)
       along the limiter segment. beta is the same for the input segment. */
      alpha = ((q_ir - p_jr)*delta_q_z - (q_iz - p_jz)*delta_q_r)/(delta_p_r*delta_q_z - delta_p_z*delta_q_r)
      beta = ((q_ir - p_jr)*delta_p_z - (q_iz - p_jz)*delta_p_r)/(delta_p_r*delta_q_z - delta_p_z*delta_q_r)
      /* For this application, one of the input points may be on a limiter segment while the other
       end crosses some other limiter segment. We are only interested in determining the latter
       since the former was arranged on purpose. The presence of the former is indicated by beta
       = 0 or 1. To exclude it, we have removed the equality tests on beta. */
      if ((alpha ≥ zero) ∧ (beta > zero) ∧ (alpha ≤ one) ∧ (beta < one)) then
        wall_cross = j
      end if
    end if
  end do
  assert(wall_cross > 0)

  return
end

```

Find point of intersection of two line segments.

```

⟨ Functions and Subroutines 2.1 ⟩ +≡
subroutine find_intersection(lines, x_int)
  implicit none_f77
  implicit none_f90

  real lines2,2,2 // Input
  real x_int2 // Output
  integer iline // Local
  logical vertical2
  real m2, b2 /* Neatest approach is to first put line segments in slope-intercept form. */
  do iline = 1, 2
    verticaliline =  $\mathcal{F}$ 
    if (abs(linesiline,2,1 - linesiline,1,1) > epsilon) then
      m_iline = (linesiline,2,2 - linesiline,1,2) / (linesiline,2,1 - linesiline,1,1)
      b_iline = (linesiline,1,2 * linesiline,2,1 - linesiline,2,2 * linesiline,1,1) / (linesiline,2,1 - linesiline,1,1)
    else
      verticaliline =  $\mathcal{T}$ 
    end if
  end do

  if ( $\neg$ vertical1  $\wedge$   $\neg$ vertical2) then
    x_int1 = (b2 - b1) / (m1 - m2)
    x_int2 = (m1 * b2 - m2 * b1) / (m1 - m2)
  else if (vertical1  $\wedge$   $\neg$ vertical2) then
    x_int1 = lines1,1,1
    x_int2 = m2 * x_int1 + b2
  else if ( $\neg$ vertical1  $\wedge$  vertical2) then
    x_int1 = lines2,1,1
    x_int2 = m1 * x_int1 + b1
  else
    assert('TwoVerticalLinesInfind_intersection' ≡ ' ')
  end if

  return
end
```

Use a Newton method to solve for the exact point on a segment having the input flux value.

```

⟨ Functions and Subroutines 2.1 ⟩ +≡
subroutine psi_solve(psi, x-seg, x-psi)

implicit none f77
implicit none f90

real psi      // Input
real x-seg2,2
real x-psi2   // Input / Output
integer i-newt  // Local
real alpha_int, psi_int, dpsi_dr, dpsi_dz, dpsi_dalpha
external psi_interp // External
real psi_interp

if (abs(x-seg2,1 - x-seg1,1) > epsilon) then
    alpha_int = (x-psi1 - x-seg1,1) / (x-seg2,1 - x-seg1,1)
else
    assert(abs(x-seg2,2 - x-seg1,2) > epsilon)
    alpha_int = (x-psi2 - x-seg1,2) / (x-seg2,2 - x-seg1,2)
end if
psi_int = psi_interp(x-psi1, x-psi2, 0, 0)
i-newt = 0 /* Use the flux gradient from psi_interp in a Newton method to refine the location of
the crossing point. */
do while((abs(psi_int - psi) > epsilon) ∧ (i-newt ≤ i-newt_max))
    dpsi_dr = psi_interp(x-psi1, x-psi2, 1, 0)
    dpsi_dz = psi_interp(x-psi1, x-psi2, 0, 1)
    dpsi_dalpha = dpsi_dr * (x-seg2,1 - x-seg1,1) + dpsi_dz * (x-seg2,2 - x-seg1,2)
    assert(abs(dpsi_dalpha) > epsilon)
    alpha_int = alpha_int + (psi - psi_int) / dpsi_dalpha
    x-psi1 = x-seg1,1 + alpha_int * (x-seg2,1 - x-seg1,1)
    x-psi2 = x-seg1,2 + alpha_int * (x-seg2,2 - x-seg1,2)
    psi_int = psi_interp(x-psi1, x-psi2, 0, 0)
    i-newt ++
end do

return
end

```

```

abs: 1.1, 2.2, 2.3, 2.4.
alpha: 2, 2.2.
alpha_int: 2.4.
aname: 1.
APP: 1.1.
areal: 1.1, 2.
arg_count: 1.1.
assert: 1, 1.1, 2, 2.2, 2.3, 2.4.
aunit: 1.

b: 2.3.
beta: 2.2.
bspline90_22: 2.

closer: 1.1.
CMOD_GPI: 1.1.

```

```
command_arg: 1.1.  
const: 1.1, 2.  
cont_ind: 1.1.  
convergence_error_count: 1.1, 2.  
  
declare_varp: 1.1.  
define_dimen: 1.1.  
define_varp: 1.1.  
definegeometry2d: 1, 1.1.  
delta_p_r: 2.2.  
delta_p_z: 2.2.  
delta_p_2: 2.2.  
delta_psi: 1.1.  
delta_q_r: 2.2.  
delta_q_z: 2.2.  
delta_q_2: 2.2.  
dim_ext: 1, 1.1.  
dim_nodes: 1, 1.1.  
dim_walls: 1, 1.1.  
direction: 1.1, 2.  
diskout: 1.1.  
dist: 1.1, 2.  
dpsi_dalpha: 2.4.  
dpsi_dr: 2.4.  
dpsi_dz: 2.4.  
dr: 1.1, 2.  
drbox: 1.1, 2.  
dzbox: 1.1.  
  
ef_common: 1.1, 2.1, 2.2.  
ef_cross_max: 1.1, 2.  
ef_lim_cross_node: 1.1, 2.  
ef_lim_cross_times: 1.1, 2.  
ef_lim_cross_wall: 1.1, 2.  
ef_lim_msegments: 1.1, 2, 2.1, 2.2.  
ef_lim_psi: 1.1.  
ef_lim_r: 1.1, 2, 2.1, 2.2.  
ef_lim_z: 1.1, 2, 2.1, 2.2.  
ef_psilim: 1.1.  
efit2dg2d: 1.1.  
epsilon: 1.1, 2.3, 2.4.  
ext_lim_ind: 1.1.  
ext_lim_index: 1, 1.1, 2.  
ext_lim_m: 1, 1.1, 2.  
ext_lim_r: 1, 1.1, 2.  
ext_lim_tempf: 1.1.  
ext_lim_tempi: 1.1.  
ext_lim_used: 1.1.  
ext_lim_wall: 1, 1.1, 2.  
ext_lim_z: 1, 1.1, 2.  
  
FALSE: 1.1.  
file: 1, 1.1.  
FILE: 1.
```

*FILELEN*: 1.1.  
*find\_intersection*: 2, 2.3.  
*finished*: 1.1, 2.  
*first\_point*: 1.1.  
*float*: 1.1.  
*FLOAT*: 1.1.  
*form*: 1.  
*fsrefine\_xgca*: 2.  
*f90*: 2.  
  
*g\_file*: 1.1.  
*Get*: 1.1.  
*get\_n\_t*: 1.  
*get\_next\_point\_tang*: 2.  
*g2\_num\_points*: 2.  
*g2\_points\_ind*: 1.1.  
*g2\_points\_ind0*: 1.1.  
*g2\_x*: 1.1, 2.  
*g2\_xz\_ind*: 1.1.  
*g2\_z*: 1.1, 2.  
  
*half*: 1.1.  
  
*i\_newt*: 2.4.  
*i\_newt\_max*: 1, 2.4.  
*ic*: 1.1.  
*iline*: 2.3.  
*im*: 1.1.  
*im\_next*: 1.1.  
*implicit\_none\_f77*: 1.1, 2.1, 2.2, 2.3, 2.4.  
*implicit\_none\_f90*: 1.1, 2.1, 2.2, 2.3, 2.4.  
*imp1*: 1.1.  
*in*: 1.  
*increment\_ext\_lim\_m*: 1, 1.1, 2.  
*increment\_num\_nodes*: 1, 1.1, 2.  
*increment\_num\_walls*: 1, 1.1.  
*init\_psi\_interp*: 1.1.  
*int*: 1.1.  
*INT*: 1.1.  
*int\_lookup*: 1.1.  
*int\_undef*: 1.1.  
*iostat*: 1.  
*ir*: 1.1.  
*is*: 1.1.  
*itr*: 1.1, 2, 2.1.  
*iwall*: 1.1.  
*iz*: 1.1.  
  
*j*: 2.2.  
*jp*: 2.2.  
  
*lim\_psi\_max*: 1.1.  
*lim\_psi\_min*: 1.1.  
*lines*: 1.1, 2, 2.3.

*m:* 2.3.  
*max:* 1, 1.1.  
*max\_err:* 1.1, 2.  
*mem\_inc:* 1.1.  
*mem\_size:* 1, 1.1.  
*mod:* 1.1, 2, 2.2.  
  
*n\_cont:* 1.1.  
*n\_cont\_in:* 1, 1.1.  
*n\_cont\_out:* 1, 1.1.  
*n\_opt:* 1, 2.  
*nargs:* 1.1.  
*next\_start:* 1.1.  
*node\_element\_count:* 1, 1.1.  
*node\_ind:* 1.1.  
*node\_type:* 1, 1.1.  
*Nodes:* 1.1.  
*nodes:* 1, 1.1, 2.  
*nrbox:* 1, 1.1.  
*nseg:* 1.1, 2.  
*nseg\_p:* 1.1, 2.  
*NSTX\_ENDD:* 1.1.  
*NSTX\_GPI:* 1.1.  
*num\_nodes:* 1, 1.1, 2.  
*num\_poly:* 1.1.  
*num\_poly\_wall:* 1.1.  
*num\_walls:* 1, 1.1, 2.  
*nzbox:* 1, 1.1.  
  
*one:* 1.1, 2.2.  
*open\_file:* 1.  
*open\_stat:* 1.  
  
*p\_jr:* 2.2.  
*p\_jz:* 2.2.  
*pointinpoly:* 2.1.  
*poly\_psi:* 1.1.  
*poly\_wall:* 1.1.  
*polygons:* 1.  
*polygons\_dg2d:* 1.  
*polygons\_done:* 1.1.  
*psi:* 1.1, 2, 2.4.  
*psi\_contour:* 1.1.  
*psi\_contours:* 1.1.  
*psi\_err:* 1.1, 2.  
*psi\_int:* 2.4.  
*psi\_interp:* 1.1, 2.4.  
*psi\_max:* 1.1.  
*psi\_min:* 1.1.  
*psi\_solve:* 1.1, 2, 2.4.  
*psi\_value:* 1.1.  
*psilim:* 1.1.  
*py:* 1.

*q\_ir:* 2.2.  
*q\_iz:* 2.2.  
*r:* 1.1, 2.1.  
*r\_dtheta:* 1, 1.1, 2.  
*rb\_ind:* 1.1.  
*rbox:* 1.1.  
*rbox\_max:* 1, 1.1.  
*rbox\_min:* 1, 1.1.  
*read\_poly:* 1.  
*r1:* 2.2.  
*r2:* 2.2.  
*segment:* 1.1.  
*sign:* 1.1.  
*sort2:* 1.1.  
*SP:* 1.1.  
*sqrт:* 1.1, 2.  
*st\_decls:* 1.1.  
**status:** 1, 1.1.  
*stratum\_psi:* 1.  
*success:* 1.1, 2.  
*sy\_decls:* 1.1.  
*test\_lim:* 2.1.  
*tracked:* 1.1.  
*TRUE:* 1.1.  
*txt:* 1.  
*unit:* 1, 1.1.  
*usr2dplasma:* 1.  
*var\_alloc:* 1.1.  
*var\_free:* 1.1.  
*var\_realloca:* 1.  
*vertical:* 2.3.  
*wall\_check:* 2, 2.1, 2.2.  
*wall\_cross:* 1.1, 2, 2.2.  
*wall\_elements:* 1, 1.1.  
*wall\_nodes:* 1, 1.1, 2.  
*wall\_psi\_contour:* 1, 1.1.  
*wall\_segment\_count:* 1, 1.1, 2.  
*wall\_unused:* 1, 1.1.  
*wall\_used:* 1.1.  
*wall\_used\_ind:* 1.1.  
*wall\_used\_once:* 1.  
*wall\_used\_twice:* 1.  
*wallfile:* 1.  
*wall2d\_ind:* 1.1.  
*web:* 1.  
*while:* 1.1, 2, 2.4.  
*x\_int:* 1.1, 2, 2.3.  
*x\_new:* 1.1, 2.  
*x\_old:* 1.1, 2.

*x\_psi*: 1.1, 2.4.  
*x\_seg*: 2.4.

*z*: 1.1, 2.1.  
*zb\_ind*: 1.1.  
*zbox*: 1.1.  
*zbox\_max*: 1, 1.1.  
*zbox\_min*: 1, 1.1.  
*zero*: 1.1, 2, 2.2.  
*z1*: 2.2.  
*z2*: 2.2.

⟨ Functions and Subroutines 2.1, 2.2, 2.3, 2.4 ⟩ Used in section 1.1.

⟨ Get Nodes 2 ⟩ Used in section 1.1.

⟨ Memory allocation interface 0 ⟩ Used in section 1.1.

**COMMAND LINE:** "fweave -f -i! -W[ -ybs15000 -ykw800 -ytw40000 -j -n/  
/Users/dstotler/degas2/src/efit2dg2d.web".

**WEB FILE:** "/Users/dstotler/degas2/src/efit2dg2d.web".

**CHANGE FILE:** (none).

**GLOBAL LANGUAGE:** FORTRAN.